AD-A245 770

ITATION PAGE

Form Approved
OPM No. 0704-0188

2

P n H M
our per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data
s burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington
efferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

1. (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED Final: 16 Nov 1991 to 01 June 1993

4. TITLE AND SUBTITLE Rational, M68020/Bare Cross-Development Facility, Version 7, R1000 Series 300 (Host) to Motorola 68020 in MVME 135 135 Board (Target), 901116W1.11083	5. FUNDING NUMBERS
6. AUTHOR(S) Wright-Patterson AFB, Dayton, OH USA	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Ada Validation Facility, Language Control Facility ASD/SCEL Bldg. 676, Rm 135 Wright-Patterson AFB, Dayton, OH 45433	8. PERFORMING ORGANIZATION REPORT NUMBER AVF-VSR-426-1290
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office United States Department of Defense Pentagon, Rm 3E114 Washington, D.C. 20301-3081	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Rational, M68020/Bare Cross-Development Facility, Version 7, Wright-Patterson AFB, R1000 Series 300 (Host) to Motorola 68020 in MVME 135 135 Board (Target), ACVC 1.11.

92-03163

14. SUBJECT TERMS Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.	15. NUMBER OF PAGES 16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION UNCLASSIFED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 16 November 1991.

Compiler Name and Version:	M68020/Bare Cross-Development Facility, Version 7
Host Computer System:	R1000 Series 300, Rational Environment Version D_12_24_0
Target Computer System:	Motorola 68020 in MVME 135 Board, Bare Machine
Customer Agreement Number:	90-07-20-RAT

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901116W1.11083 is awarded to Rational. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.

Steve P. Wilson

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

[illegible]

fr
Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

John P Solomond

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

AVF Control Number: AVF-VSR-426-1290
19 November 1991
90-07-20-RAT

Ada COMPILER VALIDATION SUMMARY REPORT:

Certificate Number: 901116W1.11083

Rational

M68020/Bare Cross-Development Facility, Version 7

R1000 Series 300 => Motorola 68020 in MVME 135 Board

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 16 November 1991.

Compiler Name and Version:	M68020/Bare Cross-Development Facility, Version 7
Host Computer System:	R1000 Series 300, Rational Environment Version D_12_24_0
Target Computer System:	Motorola 68020 in MVME 135 Board, Bare Machine
Customer Agreement Number:	90-07-20-RAT

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901116W1.11083 is awarded to Rational. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.

[illegible]

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

[illegible]

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

Customer:	Rational
Ada Validation Facility:	ASD/SCEL, Wright-Patterson AFB OH 45433-6503
ACVC Version:	1.11

Ada Implementation

Compiler Name:	M68020/ Bare Cross-Development Facility, Version 7
Host Architecture:	R1000 Series 300
Host Operating System:	Rational Environment Version D_12_24_0
Target Architecture:	Motorola 68020 in MVME 135 Board
Target Operating System:	Bare Machine

Customer's Declaration

I, the undersigned, representing Rational, declare that Rational has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration. I declare that Rational is the owner of the above implementation and the certificates shall be awarded in the name of the owners corporate name.

[illegible] Date: 7/27/90

David H. Bernstein
Vice President, Products Group

Rational
3320 Scott Blvd.
Santa Clara, CA 95054

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values — for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 12 October 1990.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
C74308A	B83022B	B83022H	B83025B	B83025D	B83026B
B85001L	C83026A	C83041A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
BD8002A	BD8004C	CD9005A	CD9005B	CDA201E	CE2107I
CE2117A	CE2117B	CE2119B	CE2205B	CE2405A	CE3111C
CE3118A	CE3411B	CE3412B	CE3607B	CE3607C	CE3607D
CE3812A	CE3814A	CE3902B			

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 21 tests check for the predefined type LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35702A, C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45624A checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 5. For this implementation, MACHINE_OVERFLOWS is TRUE.

C45624B checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 6. For this implementation, MACHINE_OVERFLOWS is TRUE.

B86001Y checks for a predefined fixed-point type other than DURATION.

C96005B checks for values of type DURATION'BASE that are outside the range of DURATION. There are no such values for this implementation.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than was specified by the length clause, as allowed by AI-00558.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

The following 265 tests check for sequential, text, and direct access files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3116A	CE3119A	EE3203A
EE3204A	CE3207A	CE3208A	CE3301A
EE3301B	CE3302A	CE3304A	CE3305A
CE3401A	CE3402A	EE3402B	CE3402C..D (2)
CE3403A..C (3)	CE3403E..F (2)	CE3404B..D (3)	CE3405A
EE3405B	CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)
CE3408A..C (3)	CE3409A	CE3409C..E (3)	EE3409F
CE3410A	CE3410C..E (3)	EE3410F	CE3411A
CE3411C	CE3412A	EE3412C	CE3413A..C (3)
CE3414A	CE3602A..D (4)	CE3603A	CE3604A..B (2)
CE3605A..E (5)	CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)
CE3705A..E (5)	CE3706D	CE3706F..G (2)	CE3804A..P (16)
CE3805A..B (2)	CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)
CE3904A..B (2)	CE3905A..C (3)	CE3905L	CE3906A..C (3)
CE3906E..F (2)			

CE2103A, CE2103B, and CE3107A use an illegal file name in an attempt to create a file and expect NAME_ERROR to be raised; this implementation does not support external files and so raises USE_ERROR. (See section 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 102 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B22003B	B22004A	B22004B	B22004C	B23002A
B23004A	B23004B	B24001A	B24001B	B24001C	B24005A
B24005B	B24007A	B24009A	B24204B	B24204C	B24204D
B25002B	B26001A	B26002A	B26005A	B28003A	B28003C
B29001A	B2A003B	B2A003C	B2A003D	B2A007A	B32103A
B33201B	B33202B	B33203B	B33301A	B33301B	B35101A
B36002A	B37106A	B37205A	B37307B	B38003A	B38003B
B38009A	B38009B	B41201A	B44001A	B44004A	B44004B
B44004C	B44004D	B44004E	B45205A	B48002A	B48002D
B53003A	B55A01A	B56001A	B63001A	B63001B	B64001B
B64006A	B67001A	B67001B	B67001C	B67001D	B67001H
B71001A	B71001G	B71001M	B74003A	B74307B	B83E01C
B83E01D	B83E01E	B91001F	B91001H	B91003E	B95001D
B95003A	B95004A	B95006A	B95007B	B95079A	BA1001B
BB3005A	BC1109A	BC1109B	BC1109C	BC1109D	BC1303F
BC2001D	BC2001E	BC3003A	BC3003B	BC3005B	BC3013A
BE2210A	BE2413A	B51001A			

CE2103A, CE2103B, and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when USE_ERROR is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

David H. Bernstein
3320 Scott Blvd.
Santa Clara CA 95054

For a point of contact for sales information about this Ada implementation system, see:

David H. Bernstein
3320 Scott Blvd.
Santa Clara CA 95054

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a) Total Number of Applicable Tests	3565
b) Total Number of Withdrawn Tests	81

c) Processed Inapplicable Tests	58	
d) Non-Processed I/O Tests	265	
e) Non-Processed Floating-Point Precision Tests	201	
f) Total Number of Inapplicable Tests	524	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

The above number of I/O tests were not processed because this implementation does not support a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 524 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation and 265 executable tests that use file operations not supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system via FTP, and run. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

Option | Switch	Effect
Create_Subprogram_Specs = False	When a library unit subprogram body is added to the program library, do not automatically create a corresponding subprogram specification.
Linker_Command_File = "!VALIDATION.ACVC_1_11.MC68020_BARE.MISCELLANY. MODIFIED_LINKER_COMMANDS"	Overrides the default linker command file with one that specifies inclusion of assembly language modules needed for pragma interface tests.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN—also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

MACRO PARAMETERS

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$MAX_IN_LEN	254
$ACC_SIZE	32
$ALIGNMENT	1
$COUNT_LAST	1000000000
$DEFAULT_MEM_SIZE	2147483647
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	MC68020_BARE
$DELTA_DOC	0.0000000004656612873077392578125
$ENTRY_ADDRESS	SYSTEM.TO_ADDRESS (16#200#)
$ENTRY_ADDRESS1	SYSTEM.TO_ADDRESS (16#201#)
$ENTRY_ADDRESS2	SYSTEM.TO_ADDRESS (16#202#)
$FIELD_LAST	2147483647
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	1.0
$GREATER_THAN_DURATION_BASE_LAST	131073.0
$GREATER_THAN_FLOAT_BASE_LAST	2.0E308
$GREATER_THAN_FLOAT_SAFE_LARGE	2#11111111111111111.1111111#E111

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E308
$HIGH_PRIORITY	255
$ILLEGAL_EXTERNAL_FILE_NAME1	BAD_CHARACTERS&<>=
$ILLEGAL_EXTERNAL_FILE_NAME2	CONTAINS_WILDCARDS*
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1
$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006F1.TST")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2147483648
$INTERFACE_LANGUAGE	ASM
$LESS_THAN_DURATION	-1.0
$LESS_THAN_DURATION_BASE_FIRST	-131073.0
$LINE_TERMINATOR	ASCII.CR
$LOW_PRIORITY	0
$MACHINE_CODE_STATEMENT	NULL;
$MACHINE_CODE_TYPE	NULL
$MANTISSA_DOC	31
$MAX_DIGITS	15
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2147483648
$MIN_INT	-2147483648

MACRO PARAMETERS

$NAME	SHORT_SHORT_INTEGER
$NAME_LIST	MC68020_BARE
$NAME_SPECIFICATION1	X2120A
$NAME_SPECIFICATION2	X2120B
$NAME_SPECIFICATION3	X3119A
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	2147483647
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	MC68020_BARE
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	NEW INTEGER
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	4096
$TICK	1.00000000000000E-03
$VARIABLE_ADDRESS	SYSTEM.TO_ADDRESS (16#00DC#)
$VARIABLE_ADDRESS1	SYSTEM.TO_ADDRESS (16#011C#)
$VARIABLE_ADDRESS2	SYSTEM.TO_ADDRESS (16#01BC#)
$YOUR_PRAGMA	NICKNAME

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

```
 PROCESSOR              SWITCH                       TYPE          VALUE

  Cross_Cg . Asm_Source                        : Boolean          := False
 -- Controls retention of assembly source code generated by the compiler.

  Cross_Cg . Auto_Download                     : Boolean          := True
 -- Controls whether the result of partially linking a main program is
 -- automatically downloaded to the target machine, using FTP switches to
 -- determine the destination. Applies only to targets that have a final
 -- link step on the target machine.

 Directory . Create_Internal_Links             : Boolean          := True
 -- Controls whether internal links are created automatically when the
 -- visible parts of library units are created. Internal links for library
 -- units are created in the set of links for the nearest enclosing world.
 -- The default is True. The full switch name is
 -- Directory.Create_Internal_Links. (For further information on links, see
 -- the Key Concepts section of the Library Management (LM) Reference
 -- Manual.) .

* Directory . Create_Subprogram_Specs          : Boolean          := False
 -- Controls whether specifications for library-unit subprograms are created
 -- automatically. The contents of these specifications are created the
 -- first time the body is successfully installed. The "with" clause for
 -- the specification is derived from the "with" clauses in the body. Only
 -- those "with" clauses required to promote the specification are included.
 --  The default is True. The full switch name is
 -- Directory.Create_Subprogram_Specs.

  Cross_Cg . Debugging_Level                   : Debug_Level      := Full
 -- Cross_Cg.Debugging_Level controls the amount of debugging assistance put
 -- into the object module when coding an Ada unit.
 --
 -- The possible values are:
```

```
 --    None    : (Default)  No debugging information produced.
 --    Partial : Debugging tables produced but optimizations are not
 -- inhibited.
 --    Full    : Debugging tables produced, and optimizations inhibited.
 --
 -- "Optimizations inhibited" means that code motion across statement
 -- boundaries will not occur, and the lifetimes of variables will not be
 -- reduced.

  Cross_Cg . Enable_Code_Pooling          : Boolean            := False
 -- When true, optimizations which would prevent link time code pooling are
 -- inhibited.  Link time code pooling is only attempted on units that were
 -- compiled with this switch set to True.

  Cross_Cg . Inlining_Level               : Inlining_Level     := Inter_Uni
 -- Cross_Cg.Inlining_Level determines how the compiler treats Pragma
 -- Inline.
 --
 -- The possible values are:
 --   None       : Ignore pragma Inline.
 --   Intra_Unit : Honor pragma Inline within a compilation unit, but ignore
 -- pragma Inline applied to subprograms in other compilation units (thus
 -- avoiding introduction of additional compilation dependencies).
 --   Inter_Unit : (Default) Honor all Inline pragmas.
 --   Full       : Honor all Inline pragmas, and additionally perform
 -- automatic inlining of small subprograms within a compilation unit.

  Cross_Cg . Linker_Command_File          : String             :=
   "!VALIDATION.ACVC_1_11.MC68020_BARE.MISCELLANY.MODIFIED_LINKER_COMMANDS"
 -- Cross_Cg.Linker_Command_File overrides the default file name for the
 -- linker command file.  The name of the linker command file is resolved in
 -- the context of the current switch file.

  Cross_Cg . Linker_Cross_Reference       : Boolean            := False
 -- Cross_Cg.Linker_Cross_Reference controls whether a cross-reference of
 -- external symbols to modules being linked is produced in the link map.

  Cross_Cg . Linker_Eliminate_Dead_Code   : Boolean            := True
 -- Cross_Cg.Linker_Eliminate_Dead_Code controls whether the linker removes
 -- unreachable subprograms from the executable program image.

  Cross_Cg . Linker_Pool_Code             : Boolean            := False
 -- Controls whether the linker eliminates redundant subprograms from the
 -- executable program image. Redundant subprograms are those that are
 -- reachable from the main program but whose code is identical to that of
 -- some other subprogram in the program.  Only comp units that were
 -- compiled with the switch Enable_Code_Pooling set to True are eligible
 -- for code pooling at link time.

  Cross_Cg . Linker_Pool_Literals         : Boolean            := True
 -- Controls whether the linker eliminates redundant literals from the
 -- executable program image.
 Cross_Cg . Listing                     : Boolean               := False
-- Controls generation of machine code listing file.

 Cross_Cg . Optimization_Level          : Integer range 0 .. 3 := 3
-- Cross_Cg.Optimization_Level controls the amount of optimization
-- performed during code generation.
--
-- The possible values are:
--    0 : Minimal Optimization
--    1 : Unimplemented
--    2 : Unimplemented
--    3 : Maximal Optimization.

Directory . Require_Internal_Links      : Boolean               := True
-- Controls whether failure to create internal links (as controlled by the
-- Directory.Create_Internal_Links switch) generates an error.  The default
-- (True) is to treat the failure to generate links as an error and to
-- discontinue the operation.  If the Directory.Create_Internal_Links
-- switch is set to False, this switch has no effect.  The full switch name
-- is Directory.Require_Internal_Links.

 Cross_Cg . Suppress_All_Checks         : Boolean               := False
-- When true, this switch has the same effect as a pragma Suppress_All at
-- the beginning of the each Ada unit in the library.

 Cross_Cg . Target_Linker_Script        : String                := ""
-- Cross_Cg.Target_Linker_Script overrides the default file name for the
-- target linker script file.  This name of this file is resolved in the
-- context of the current switch file. This switch applies only to targets
-- having a final link step on the target machine.
```

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is
    ..........
    type Integer is range -2147483648 .. 2147483647;
    type Short_Short_Integer is range -128 .. 127;
    type Short_Integer is range -32768 .. 32767;

    type Float is digits 6 range -16#1.FFFF_FE# * 2.0 ** 127 ..
                                  16#1.FFFF_FE# * 2.0 ** 127;

    type Long_Float is digits 15 range
            -16#1.FFFF_FFFF_FFFF_F# * 2.0 ** 1023 ..
             16#1.FFFF_FFFF_FFFF_F# * 2.0 ** 1023;

    type Duration is delta 16#1.0# * 2.0 ** (-14)
                    range -16#1.0# * 2.0 ** 17 ..
                           16#1.FFFF_FFFC# * 2.0 ** 16;

    ..........
end STANDARD;
```

Appendix IV: Appendix F to the LRM for the M68020/Bare Target

Appendix F describes the implementation-dependent features of the Ada language, as it is implemented for the M68020/Bare target. If you are using a CDF to compile programs for other targets, refer to the Appendix F that is provided with the documentation for that CDF. If you are compiling programs for an R1000 target, refer to the Appendix F for the R1000 target.

Appendix F is a required part of the *Reference Manual for the Ada Programming Language* (LRM) and is divided into the sections listed below:

- "Implementation-Dependent Pragmas" describes the form, allowed places, and effect of every implementation-dependent pragma.
- "Implementation-Dependent Attributes" describes the name and type of every implementation-dependent attribute.
- "Packages Standard and System" presents the specifications of packages Standard and System.
- "Support for Representation Clauses" lists all of the restrictions on representation clauses.
- "Implementation-Generated Names" describes the conventions used for any implementation-generated name denoting implementation-dependent components of records.
- "Address Clauses" describes the interpretation of expressions that appear in address clauses, including those for interrupts.
- "Unchecked Programming" describes any restrictions on unchecked conversions and deallocations.
- "Input/Output Packages" describes implementation-dependent characteristics of the input/output packages.
- "Other Implementation-Dependent Features" describes implementation-dependent features not covered in the previous sections.

IMPLEMENTATION-DEPENDENT PRAGMAS

The CDF accepts the pragmas defined in Annex B of the LRM, as well as a number of additional pragmas to be used in application software development. The first of the following subsections lists clarifications and restrictions for the pragmas defined in Annex B.

For each of the pragmas defined for the M68020 Family CDF, the following subsections describe the extent to which it is supported for the M68020/Bare target. Support for these pragmas may differ for other targets. Information about the pragmas that are supported for each target is given in the CDF manual for that target.

Pragmas Defined in Annex B

For each of the pragmas defined in Annex B of the LRM, Table IV-1 describes the extent to which it is supported for the M68020 family target. Support for these pragmas may differ for other targets.

Table IV-1 Predefined Pragmas

Pragma	Effect
Controlled	Always implicity in effect because the implementation does not support automatic garbage collection.
Elaborate	As given in Annex B of the LRM.
Inline	As given in Annex B of the LRM, subject to the setting of the switch Inlining_Level.
Interface	Used in conjunction with pragmas Import_Procedure, Import_Valued_Procedure, and Import_Function.
List	As given in Annex B of the LRM; evident only when the Compile command is used.
Memory_Size	The pragma has no effect.
Optimize	Has no effect.
Pack	Removes gaps in storage, minimizing space with possible increase in access time. See the "Size of Objects" subsection in the "Support for Representation Clauses" section.
Page	As given in Annex B of the LRM; evident only when the Compile command is used.
Priority	As given in Annex B of the LRM.
Shared	As given in Annex B of the LRM; has an effect only for integer, enumeration, access, and fixed types.
Storage_Unit	Has no effect.
Suppress	As given in Annex B of the LRM.
System_Name	Has no effect. (There is only one enumeration literal in the type System.System_Name

Pragma Export_Elaboration_Procedure

Makes the elaboration procedure for a given compilation unit available to external code by defining a global symbolic name.

Format

```
Pragma Export_Elaboration_Procedure ( External_Name => "external_name" );
```

Description

The elaboration procedure is otherwise unnameable by the user. Its use is confined to the exceptional circumstances where an Ada module is not elaborated because it is not in the closure of the main program or the main program is not an Ada program. This pragma is not recommended for use in application programs unless the user has a thorough understanding of elaboration, runtime, and storage-model considerations.

Pragma Export_Elaboration_Procedure must appear immediately following the compilation unit. The external name is a string literal.

Pragmas Export_Object and Import_Object

Imports or exports objects from an Ada unit.

Format

```
pragma object_pragma_type
                         ( Internal => internal_identifier,
                           External => "external_name" ) ;
```

Description

The Import_Object pragma causes an Ada name to reference storage declared and allocated in some external (non-Ada) object module. The Export_Object pragma provides an object declared within an Ada unit with an external symbolic name that the linker can use to allow another program to access the object. It is the responsibility of the programmer to ensure that the internal structure of the object and the assumptions made by the importing code and data structures correspond. The cross-compiler cannot check for such correspondence.

Note: The object to be imported or exported must be a variable declared at the outermost level of a library-package specification or body. The pragmas must be in the same declarative part as the variable.

The size of the object must be static. Thus, the type of the object must be one of:

- A scalar type (or subtype)
- An array subtype with static index constraints whose component size is static
- A nondiscriminated record type or subtype

Objects of a private or limited private type can be imported or exported only into the package that declares the type.

An imported object cannot have an initial value and thus cannot be:

- Declared with the keyword *constant*
- An access type
- A record type with discriminants
- A record type whose components have default initial expressions
- A record or array whose components contain access types or task types

In addition, the object must not be in a generic unit. The external name specified must be suitable as an identifier in the assembler.

Parameters

- ***object_pragma_type***: Valid values are either Import_Object or Export_Object.
- **Internal => *internal_identifier***: Specifies the Ada (internal) name of the object. This parameter is required.
- **External => *"external_name"***: Specifies the external symbolic name of the object. This parameter is optional; if not specified, the internal name is used.

Pragmas to Import and Export Functions and Procedures

Allows Ada subprograms to be used by non-Ada routines, and vice-versa.

Format

```
pragma interface_type
                        ( Internal => internal_name,
                          External =>  external_name,
                          Parameter_Types =>  parameter_type_list,
                          Result_Type => type_mark,
                          Nickname => "nickname",
                          Mechanism => mechanism_list,
                          Language => language_name );
```

Parameters

- *interface_type:* Valid values are:

  — Import_Procedure

  — Import_Function

  — Import_Valued_Procedure

  — Export_Procedure

  — Export_Function

- **Internal => *internal_name***: Designates the Ada name of the subprogram being interfaced. The internal name can be either an identifier or a string literal. If more than one subprogram is in the declarative region preceding the exporting pragma, the correct subprogram must be identified by either using the Parameter_Types (and Result_Type, if a function) or specifying the nickname with pragma Nickname and the Nickname argument or both.
- **External => *external_name***: Specifies the name to be used by the assembler. This is a character string that is an identifier suitable for the M68020/Bare assembler. If the external designator is not specified, the internal name is used.
- **Parameter_Types => *parameter_type_list***: Distinguishes among two or more overloaded subprograms having the same internal name. The value of the Parameter_Types argument is a list of type or subtype names separated by commas and enclosed in parentheses. Each name corresponds positionally to a formal parameter in the subprogram's declaration. If the subprogram has no parameters, the list consists of the single word Null.
- **Result_Type => *type_mark***: Specifies the type returned by the function. The Result_Type argument serves the same purpose for the return values of functions as the Parameter_Type argument serves for the parameter list.
- **Nickname => *"nickname"***: See pragma Nickname.
- **Mechanism => *mechanism_list***: Specifies, in a parenthesized list, the passing mechanism for each parameter to be passed. The Mechanism argument is required if an imported subprogram has any parameters; it cannot be used for an exported subprogram. A mechanism must be specified for each parameter listed in **parameter_types** and they must correspond positionally. The types of mechanism are as follows:

  — Value: Specifies that the parameter is passed on the stack by value.

  — Reference: Specifies that the address of the parameter is passed on the stack.

  For functions, it is not possible to specify the passing mechanism of the function result; the standard Ada mechanism for the given type of the function result must be used by the interfaced subprogram. If there are parameters, and if they all use the same passing mechanism, an alternate form for the Mechanism argument can be used: instead of a parenthesized list with an element for each parameter, the single mechanism name (not parenthesized) can be used.
- **Language => *language_name***: Specifies the name of the language to which the subprogram is being exported for the sole purpose of controlling the manner in which parameters are popped from the stack. The Language argument is optional for exporting routines and cannot be used for importing routines. Any language identifier other than **C** is ignored. If the language is C, or if the subprogram has copy-back parameters (*in-out* and *out* scalars and access types), then the exported subprogram will not pop its parameters; otherwise it will.

Exporting Subprograms

A subprogram written in Ada can be made accessible to code written in another language by using an exporting pragma defined by the M68020 family cross-compiler. The effect of such a pragma is to give the subprogram a global symbolic name that the linker can use when resolving references between object modules.

Exporting pragmas can be applied only to nongeneric procedures and functions.

An exporting pragma can be given only for a subprogram that is a library unit or that is declared in the specification of a library package. An exporting pragma can be placed after a subprogram body only if the subprogram does not have a separate specification; if it has a separate specification, the pragma must go there.

Importing Subprograms

A subprogram written in another language (typically, assembly language) can be called from an Ada program if it is declared with a pragma Interface. Every interfaced subprogram must have an importing pragma that is recognized by the M68020 family cross-compiler— either Import_Procedure, Import_Valued_Procedure, or Import_Function. These pragmas are used to declare the external name of the subprogram and the parameter-passing mechanism for the subprogram call. If an interfaced subprogram does not have an importing pragma, or if the importing pragma is incorrect, pragma Interface is ignored.

Importing pragmas can be applied only to nongeneric procedures and functions.

Import_Procedure calls a non-Ada procedure; Import_Function, a non-Ada function; and Import_Valued_Procedure, a non-Ada function containing the equivalent of *out* or *in out* parameters.

Each import pragma must be preceded by a pragma Interface; otherwise, the placement rules for these pragmas are identical to those of the pragma Interface given in Section 13.9 of the LRM.

Importing Functions with In-Out Parameters

The third pragma is provided because the Ada language allows functions to have only *in* parameters. A non-Ada function containing parameters whose values can be altered by the function must be associated with an Ada procedure using Import_Valued_Procedure. The first parameter in the Ada procedure corresponds to the non-Ada function result; it must be of mode *out* and of a discrete type.

C-Language Routines

If the language C is specified in pragma Interface for an imported subprogram, the compiler assumes that the imported subprogram will not pop its parameters. If any other language is given, the compiler assumes that the imported subprogram will pop its parameters if it has no copy-back parameters (*in-out* and *out* scalars and access types), and that it won't pop its parameters otherwise.

Effect of Exporting and Importing on Elaboration Checks

Exporting a subprogram does not export the mechanism used by the compiler to perform elaboration checks; calls from other languages to an exported subprogram whose body is not elaborated may have unpredictable results when the subprogram body references objects that are not yet elaborated. Elaboration checks within the Ada program are not affected by the exporting pragma.

Use of the Import_Procedure pragma for a subprogram guarantees that no elaboration check is performed on the imported procedure. Hence, no explicit suppress of elaboration check is needed.

Examples

```
procedure Matrix_Multiply (A, B: in Matrix; C: out Matrix);

pragma Export_Procedure (Matrix_Multiply);
-- External name is the string "Matrix_Multiply"

function Sin (R: Radians) return Float;
pragma Export_Function
           (Internal => Sin,
            External => "SIN_RADIANS");
            -- External name is the string "SIN_RADIANS"

procedure Locate (Source : in String;
                  Target : in String;
                  Index  : out Natural);

pragma Interface (Assembler, Locate);
pragma Import_Procedure
           (Internal        => Locate,
            External        => "STR$LOCATE",
            Parameter_Types => (String, String, Natural),
            Mechanism       => (Reference, Reference, Value));


function Pwr (I: Integer; N: Integer) return Float;
function Pwr (F: Float; N: Integer) return Float;

pragma Interface (Assembler, Pwr);

pragma Import_Function
           (Internal        => Pwr,
            Parameter_Types => (Integer, Integer),
            Result_Type     => Float,
            Mechanism       => Value,
            External        => "MATH$PWR_OF_INTEGER");

pragma Import_Function
           (Internal        => Pwr,
            Parameter_Types => (Float, Integer),
            Result_Type     => Float,
            Mechanism       => Value,
            External        => "MATH$PWR_OF_FLOAT");
```

Pragma Interrupt_Handler

Provides a method for interrupt-handling.

Format

See examples below.

Description

Three different mechanisms are available to support interrupt handling: address clauses on task entries, subprograms identified with pragma Interrupt_Handler, and interrupt-handling queues that employ both subprograms and task entries.

- Simple interrupt handling can be accomplished with address clauses attached to task entries, as described in the LRM (Section 13.5.1). Note that the task entry must always be available. discusses this method in more detail.
- As one alternative, interrupt-handling procedures can be called on a nonspecific, target-dependent thread. In this method, the Interrupt_Handler pragma associates an interrupt-handling procedure with a corresponding interrupt vector. The syntax for the pragma is as follows:

```
procedure Handler_Procedure (Target_Dependent_Parameter : Integer);

pragma Interrupt_Handler (Handler => Handler_Procedure,
                          Vector  => [address-expression]);
```

- The third alternative for interrupt handling is the queued interrupt handler, a combination of the first two approaches. In this method, a procedure, a task entry, and an interrupt vector are associated with each other through an Interrupt_Handler pragma as shown in this example of a task specification:

```
task type Driver is
  entry Handler (Target_Dependent_Parameter : integer);
end Driver;

T : Driver;

procedure My_Interrupt_Handler (Target_Dependent_Parameter : Integer);

pragma Interrupt_Handler (Handler    => My_Interrupt_Handler,
                          Vector     => [address-expression],
                          Task_Entry => T.Handler);
```

  Note that an address clause must not be included on the entry in the task specification.

Chapter 9 discusses the development of interrupt handlers.

More than one Interrupt_Handler pragma can be associated with a given subprogram and/or task entry if that subprogram and/or task entry are to serve as handlers for more than one interrupt vector. The elaboration of the Interrupt_Handler pragma has the effect of associating either a task entry or a subprogram with an interrupt vector. This may result in the propagation of the Standard.Program_Error exception if the vector already has an associated handler.

The cross-compiler verifies that all associated handlers and named task objects are declared at the outermost scope. The pragma must appear in the same declarative region or package specification as, and following, the definitions of the task entry and the subprogram.

The interrupt-handling procedure must have a single formal parameter of type Standard.Integer. The actual value of this parameter and interpretation of it during execution of the handler is target-dependent. The Vector parameter is interpreted by the runtime system.

When an interrupt occurs, the associated procedure is called directly by the runtime system at interrupt level on an interrupt stack. No elaboration check is performed, even if elaboration checks are enabled. The context at the time of the call is the context at the time of the interrupt, not the context of the associated task, if there is one.

The interrupt-handling subprogram is responsible for clearing the interrupt and performing any device-specific actions required. The

The procedure—and all subprograms called from this procedure—must conform to a set of restrictions that include the following:

- No stack checks—use pragma Suppress (Storage_Check)
- Must not raise exceptions (ever!)
- Must not perform any tasking-related operations, including but not limited to:
  - — Entry calls
  - — Task creation
  - — Abort statements
  - — Delay statements
- Must not pop its parameters from the stack.

Failure to comply with these rules makes the program erroneous. The procedure may make use of the Runtime_Interface package to control various aspects of certain tasks. The predefined package Calendar can be used and dynamic memory allocation/deallocation is allowed. No checks are performed to ensure that restrictions are not violated, and such violations may have unpredictable results.

When the interrupt-handling subprogram returns, the runtime system checks whether the interrupt has an associated task entry. If so, the runtime system queues a call to the associated task entry in such a way that it takes precedence over any noninterrupt-driven calls to the same task entry. The task object must be activated before receiving the first interrupt; no check is performed at run time to ensure that this has been done. The interrupts are fully buffered and the called task accepts one entry call for each interrupt regardless of the rate at which interrupts are received. For example, if ten interrupts are received before the task accepts the first, then the 'Count of the associated entry is 10, and ten accept statements for that entry are required to reduce the 'Count to 0. In other words, when an accept statement is encountered, one of the following occurs:

- If the 'Count of pending interrupt-driven calls to the appropriate entry is nonzero, 'Count is decremented and execution is continued as if an accept has occurred.
- If 'Count is 0 and there are pending calls to the appropriate entry from noninterrupt-driven tasks, the first one in the queue is accepted.
- If 'Count is 0 and there are no other pending entry calls, execution is suspended, awaiting one of the above to occur.

In the situation shown in Figure IV-1, three tasks (T1, T2, and T3) have issued normal entry calls to task T's Entry1, two (T4 and T5) have issued calls to Entry2, and one (T6) has issued a call to Entry3. In addition, three interrupts have issued calls to Entry1 and one interrupt has issued a call to Entry3.

Figure IV-1 Queued Interrupts

If T'Body contains the following code:

```
. . .
begin loop
   accept Entry1;
   . . .
   accept Entry2;
   . . .
   accept Entry3;
   . . .
end loop;
```

then the first time through the loop, **accept Entry1** decrements 'Count (leaving it at 2) and continues, **accept Entry2** accepts the call from task T4, and **accept Entry3** decrements its 'Count (leaving it at 0) and continues. The second time through, **accept Entry1** again decrements 'Count, **accept Entry2** accepts the call from task T5, and **accept Entry3** accepts the call from task T6.

The priority of the task during the rendezvous is proportional to the priority of the interrupt and higher than Standard.System.Priority'Last.

discusses the development of interrupt handlers.

Pragma Main

Designates an Ada main unit.

Format

```
pragma Main ( Target     => simple_name,
              Stack_Size => static_integer_expression,
              Heap_Size  => static_integer_expression );
```

Parameters

Pragma Main has three optional parameters:

- **Target => *simple_name***: Specifies the target key as a string. If this parameter appears and does not match the current target key, pragma Main is ignored. If the Target parameter matches the current target key or does not appear, pragma Main is honored. A single source copy of a main program can be used for different targets by putting in multiple Main pragmas with different Target parameters and different stack sizes and/or different heap sizes.
- **Stack_Size => *static_integer_expression***: Specifies the size in bytes of the main task stack as an expression. If not specified, the default value is 4 Kb.
- **Heap_Size => *static_integer_expression***: Specifies the size in bytes of the heap as an expression. If not specified, the default value is 64 Kb.

Description

A parameterless library-unit procedure without subunits can be designated as a main program by including a pragma Main at the end of the unit specification or body. This pragma causes the linker to run and create an executable program when the body of this subprogram is coded. Before a unit having a pragma Main can be coded, all units in the *with* closure of the unit must be coded.

Multiple Main pragmas can be placed in the specification, the body, or both. If more than one pragma Main is specified with the same target parameter, only the first pragma Main in the specification (if there is one) has any effect; otherwise, only the first one in the body is used.

Using the Target parameter forces the pragma to be ignored for all targets but the one specified. This enables joined views of a procedure to have different effects according to the target. One use is to avoid the effects of declaring a pragma Main when the target is the R1000:

```
pragma Main (Target => Mc68020_Bare);
```

Another use is to specify different stack or heap sizes for different targets. For example:

```
procedure Show_Pragma_Main is
begin
    Do_Something;
end Show_Pragma_Main;
pragma Main (Target => Mc68020_Bare, Heap_Size => 10*1024);
pragma Main (Target => <another target key>, Heap_Size => 20*1024);
```

The program Show_Pragma_Main will be a main program in both an Mc68020_Bare view and a view for the other target. The heap sizes for the two targets will be as specified by the different Main pragmas.

Pragma Nickname

Gives a unique string name to a procedure or function in addition to its normal Ada name.

Format

```
pragma Nickname ("string");
```

Description

This unique name can be used to distinguish among over- loaded procedures or functions in the importing and exporting pragmas defined in preceding subsections.

Pragma Nickname must appear immediately following the declaration for which it is to provide a nickname. It has a single argument, the nickname, which must be a string constant.

For example:

```
function Cat (L: Integer; R: String) return String;
pragma Nickname ("Int-Str-Cat");

function Cat (L: String; R: Integer) return String;
pragma Nickname ("Str-Int-Cat");

pragma Interface (Assembly, Cat);

pragma Import_Function (Internal  => Cat,
                        Nickname  => "Int-Str-Cat",
                        External  => "CAT$INT_STR_CONCAT",
                        Mechanism => (Value, Reference));

pragma Import_Function (Internal  => Cat,
                        Nickname  => "Str-Int-Cat",
                        External  => "CAT$STR_INT_CONCAT",
                        Mechanism => (Reference, Value));
```

Pragma Suppress_All

Duplicates the effect of several Suppress pragmas.

Format

```
pragma Suppress_All;
```

Description

Pragma Suppress_All is equivalent to the following sequence of Suppress pragmas. It has no effect in a package specification. See the LRM, section 11.7.3.

```
pragma Suppress (Access_Check);
pragma Suppress (Discriminant_Check);
pragma Suppress (Division_Check);
pragma Suppress (Elaboration_Check);
pragma Suppress (Index_Check);
pragma Suppress (Length_Check);
pragma Suppress (Overflow_Check);
pragma Suppress (Range_Check);
pragma Suppress (Storage_Check);
```

Note that, like pragma Suppress, pragma Suppress_All does not prevent the raising of certain exceptions. For example, numeric overflow or dividing by zero is detected by the hardware, which results in the predefined exception Numeric_Error. Refer to , "Runtime Organization," for more information.

Pragma Suppress_All must appear immediately within a declarative part.

IMPLEMENTATION-DEPENDENT ATTRIBUTES

There are no implementation-dependent attributes.

PACKAGES SYSTEM AND STANDARD

Package Standard (LRM Annex C)

Package Standard defines all the predefined identifiers in the language.

```
package Standard is

    type *Universal_Integer* is ...
    type *Universal_Real*    is ...
    type *Universal_Fixed*   is ...
    type Boolean             is (False, True);
    type Integer             is range -2147483648 .. 2147483647;
    type Short_Short_Integer is range -128 .. 127;
    type Short_Integer       is range -32768 .. 32767;

    type Float      is digits 6 range -16#1.FFFF_FE# * 2.0 ** 127 ..
                                  16#1.FFFF_FE# * 2.0 ** 127;
    type Long_Float is digits 15 range -16#1.FFFF_FFFF_FFFF_F# * 2.0 ** 1023
                                     .. 16#1.FFFF_FFFF_FFFF_F# * 2.0 ** 1023;

    type Duration   is delta 16#1.0# * 2.0 ** (-14)
                        range -16#1.0# * 2.0 ** 17 ..
                               16#1.FFFF_FFFC# * 2.0 ** 16;

    subtype Natural  is Integer range 0 .. 2147483647;
    subtype Positive is Integer range 1 .. 2147483647;
    type Character is ...
    type String is array (Positive range <>) of Character;
    pragma Pack (String);

    package Ascii is ...

    Constraint_Error : exception;
    Numeric_Error    : exception;
    Storage_Error    : exception;
    Tasking_Error    : exception;
    Program_Error    : exception;

end Standard;
```

Table IV-2 shows the default integer and floating-point types:

Table IV-2 Supported Integer and Floating-Point Types

Ada Type Name	Size
Short_Short_Integer	8 bits
Short_Integer	16 bits
Integer	32 bits
Float	32 bits
Long_Float	64 bits

Fixed-point types are implemented using the smallest discrete type possible; it may be 8, 16, or 32 bits.

Standard.Duration is 32 bits.

Package System (LRM 13.7)

```
package  System is

    type Address is private;

    type Name is (Mc68020_Bare);

    System_Name  : constant Name := Mc68020_Bare;
    Storage_Unit : constant := 8;
    Memory_Size  : constant := +(2 ** 31) - 1;

    Min_Int : constant := -(2 ** 31);
    Max_Int : constant := +(2 ** 31) - 1;

    Max_Digits   : constant := 15;
    Max_Mantissa : constant := 31;
    Fine_Delta   : constant := 1.0 / (2.0 ** 31);
        Tick         : constant := 1.0 / 1000.0;

    subtype Priority is Integer range 0 .. 255;

    function To_Address (Value : Integer) return Address;
    function To_Integer (Value : Address) return Integer;
    function "+" (Left : Address; Right : Integer) return Address;
    function "+" (Left : Integer; Right : Address) return Address;
    function "-" (Left : Address; Right : Address) return Integer;
    function "-" (Left : Address; Right : Integer) return Address;

    function "<"  (Left, Right : Address) return Boolean;
    function "<=" (Left, Right : Address) return Boolean;
    function ">"  (Left, Right : Address) return Boolean;
    function ">=" (Left, Right : Address) return Boolean;
    --
    --  The functions above are unsigned in nature. Neither Numeric_Error
    --  nor Constraint_Error will ever be propagated by these functions.
    --
    --  Note that this implies:
    --
    --            To_Address (Integer'First) > To_Address (Integer'Last)
    --
    --  and that:
    --
    --            To_Address (0) < To_Address (-1)
    --
    --  Also, the unsigned range of Address includes values that are
    --  larger than those implied by Memory_Size.
    --

    Address_Zero : constant Address;

private

    . . .

end System;
```

SUPPORT FOR REPRESENTATION CLAUSES

The M68020/Bare CDF support for representation clauses is described in this section with references to relevant sections of the LRM. Use of a clause that is unsupported as specified in this section or use contrary to LRM specification will cause a semantic error unless specifically noted. Further details on the effects on specific types of objects are given in the "Size of Objects" subsection.

Length Clauses (LRM 13.2)

Length clauses are supported by the M68020 Family CDF as follows:

- The value of a 'Size attribute must be a positive static integer expression. 'Size attributes are supported for all scalar and composite types with the following restrictions:
  - — For all types, the value of the 'Size attribute must be greater than or equal to the minimum size necessary to store the largest possible value of the type.
  - — For discrete types, the value of the 'Size attribute must be less than or equal to 32.
  - — For fixed types, the value of the 'Size attribute must be less than or equal to 32.
  - — For float types, the 'Size attribute can specify only the size the type would have if there were no clause. The only possible legal values therefore are 32 and 64.
  - — For access and task types, the value of the 'Size attribute must be 32.
  - — For composite types, a size specification must not imply compression of composite components. Such compression must have been explicitly requested using a length clause or pragma Pack on the component type.
- 'Storage_Size attributes are supported for access and task types. The value given by a 'Storage_Size attribute can be any integer expression, and it is not required to be static.
- 'Small attributes are supported for fixed point types. The value given by a 'Small attribute must be a positive static real number that cabnot be greater than the delta of the base type. It need not be a power of 2.

Enumeration Representation Clauses (LRM 13.3)

Enumeration representation clauses are supported with the following restrictions:

- The allowable values for an enumeration clause range from (Integer'First + 1) to Integer'Last.

Record Representation Clauses (LRM 13.4)

Both full and partial representation clauses are supported for both discriminated and undiscriminated records. Record component clauses are not allowed on:

- Array or record fields whose constraint involves a discriminant of the enclosing record
- array or record fields whose constraint is not static

The static simple expression in the alignment clause part of a record representation clause—see LRM 13.4 (4)—must be a power of 2 with the following limits:

```
   1 <= static_simple_expression <= 16
```

The size specified for a discrete field in a component clause should not exceed 32 bits.

Change of Representation (LRM 13.6)

Change of representation is supported wherever it is implied by support for representation specifications.

Size Of Objects

This section describes the size of both scalar and composite objects. The first two subsections cover concepts of size that apply to all object types. The remaining subsections cover individual types. The size concepts are most important for the composite types.

Minimum, Default, Packed, and Unpacked Sizes

The following terms are used to describe the size of an object:

- Storage unit: Smallest addressable memory unit. The size of the storage unit in bits is given by the named number System.Storage_Unit. Since the MC68020 is byte-addressable, the size of the storage unit is 8.
- Minimum size for a type: The minimum number of bits required to store the largest value of the type. For example, the minimum size of a Boolean is 1.
- Maximum size for a type: The largest allowable size for a *discrete* type. For the MC68020, the maximum size is 32.
- Packed size for a type: The size of a component used in an array or record when a pragma Pack is in effect. This is the same as the minimum size unless modified by a 'Size clause (see "Determination of Storage Size," below).
- Unpacked size for a type: The size of a component used in an array or record when *no* pragma Pack is in effect. This is the same as the default size unless modified by a 'Size clause (see "Determination of Storage Size," below).
- Default size for a type: The smallest number of bits required to store the largest value of the type *when stored in whole storage units*. For composite types, the default sizes are multiples of 8. The possible default sizes for scalar, access and task types are given in Table IV-3.

Table IV-3 Default Sizes for Scalar, Access, and Task Types

Type	Sizes in Bits
Discrete	8, 16, 32
Fixed	8, 16, 32
Float	32, 64
Access	32
Task	32

Determination of Storage Size

Top-level scalar and access objects are stored using their unpacked size (a *top-level object* is an object that is not a component of any array or record). Components of composite objects having neither pragma Pack nor a record representation clause are also stored using the unpacked size. Components of composite objects having pragma Pack are stored using the packed size. Fields of records having record representation clauses can be stored in any number of bits ranging from the minimum size to the default size of the field type. If a scalar or composite type component field is specified to be smaller than the default size, a *filler field* is introduced, and the data is left justified. For further information, see the "Array Types" and "Record Types" subsections.

Discrete Types

'Size clauses on discrete types affect sizes by changing the packed and unpacked sizes. When there is no 'Size clause, the packed and unpacked sizes are the minimum and default sizes, respectively. 'Size clauses with values outside the minimum and maximum sizes cause a semantic error. Within that range, there are two cases depending on the value specified by the clause:

- *Value* <= default size: The packed size is set equal to *Value*. The unpacked size is not affected.
- *Value* > default size. The packed size is set equal to *Value* and the unpacked size is set to the number of bits in the smallest number of storage units that will hold the packed size.

Examples are provided in Table IV-4.

Table IV-4 Size Examples for M68020/Bare Target

Example Type Declaration	Minimum Size	Default Size	Maximum Size
Integer	32	32	32
Boolean	1	8	32
Float	32	32	32
type Byte is range 0 .. 255	8	16	32
type Primary is (Red, Blue, Yellow)	2	8	32
type X is (Normal, Read_Error, Write_Error) for X use (7, 15, 31)	5	8	32
type Ary is array (1 .. 100) of Boolean	100	800	n/a

Integer Types

A first-named integer subtype has a default size the same as that of the smallest integer type defined in package Standard that will hold the range chosen. For example, consider the following type declaration:

```
type Byte is range 0..255;
```

Type Byte will have a minimum size of 8 and a default size of 16. It has a default size of 16 because the smallest type from which Byte can be derived is Short_Integer. (Short_Short_Integer, which has a size of 8, does not include values greater than 127.)

The 'Size clause is supported for nonderived and derived integer types. The effect of a 'Size clause on minimum size is shown in the following example. Consider:

```
type Byte is range 0 .. 255;
for Byte'Size use n;
```

where *n* is a static integer expression. Table IV-5 shows the effect of *n* on the packed and unpacked sizes.

Table IV-5 Example of Effect of 'Size Clauses

'Size clause	Packed Size	Unpacked Size
No 'Size clause	8	16
Use 8	8	8
Use 12	12	16
Use 16	16	16

Enumeration Types

For an enumeration type with *n* elements, the default internal integer representation ranges from 0 .. *n* – 1. The maximum number of elements that can be declared for any one enumeration type depends on the total number of characters in the images of the enumeration literals. Let *L* be the total number of characters of the *n* elements. Then *L* and *n* must satisfy the following inequality: $2*n + 4 + L <= 2^{16}$

Enumeration and 'Size clauses are permitted on derived types. However, this may generate additional code when parent/derived types are converted to each other.

For predefined type Character, the value returned by the 'Size attribute is 8 and the minimum size is 8. User-defined character types behave like ordinary enumeration types and may have a minimum size that is less than 8.

The 'Size clause is supported for both nonderived and derived enumeration types. The effect of a 'Size clause on representation is shown in the following example. Consider:

```
   type Response is (No, Maybe, Yes);
   for Response'Size use n;
```

where *n* is a static integer expression. Table IV-6 lists the packed and unpacked sizes for different values of *n*.

Table IV-6 Example of Enumeration Type Sizes

'Size clause	Packed Size	Unpacked Size
No 'Size clause	2	8
Use 4	4	8
Use 12	12	16
Use 16	16	16
Use 20	20	32
Use 32	32	32

Floating-Point Types

The internal representations for floating-point types are the 32-bit and 64-bit floating-point representations as defined by the MC68020 architecture.

Fixed-Point Types

Fixed-point types are represented internally as integers. The integer representation is computed by scaling (dividing) the fixed-point number by the *actual small* implied by the fixed-point type declaration. Actual small is defined to be the nearest multiple of 2 that will represent the smallest possible value of the fixed-point type. The values that are exactly representable are those that are precise multiples of the actual small; numbers between those values are represented by the closest exact multiple. For example, in the declaration:

```
type fix is delta 0.01 range -10.0 .. 10.0;
```

the integer value used to represent the lower bound of the type is −10.0 / (1/128), or −1280, since the actual small, representing 0.01, is 1/128. In the example:

```
type fix is delta 0.01 range -10.6 .. 10.6;
```

the integer value used to represent the lower bound of the type is −1357, which is the closest exact multiple of the actual small. (This is −10.6/(1/128) = −1356.8; the nearest integer representation is −1357.)

The size of the representation is 32, 16, or 8 bits; the compiler chooses the smallest of these that can represent all of the safe numbers of the fixed-point type.

'Size and 'Small are supported for both nonderived and derived types. The value given in a 'Small clause for a fixed type must be a positive static real number. The value need not be a power of 2. By Ada rules, it may not be greater than the delta of the base type.

Access Types And Task Types

Access and task objects have a size of 32 bits. The 'Storage_Size length clause is allowed for access and task types. The value given in a Storage_Size clause can be any integer expression, and it is not required to be static. Static expressions larger than Integer'Last will generate compilation warnings; however, a Numeric_Error exception will be raised at run time. For access types, the 'Storage_Size clause is used to specify the size of the access type's collection. If a 'Storage_Size clause has been applied to an access type, the collection is nonextensible. For task types, the clause determines the stack size.

A value (either static or not) of 0 is allowed; in this case, no collection or task stack space will be allocated, and a Storage_Error exception will be raised at run time if any attempt is made to allocate or deallocate from the collection or activate the task. Negative values are also allowed by the CDF; however, this will generate a Storage_Error exception when the type is elaborated even if no attempt is made to allocate or deallocate objects belonging to the collection.

Array Types

For a given array type, the CDF is capable of using one of two representations, known as the *unpacked* and *packed* representations.

In the unpacked representation, each array component starts on a storage-unit boundary, and filler (of up to three storage units) may be introduced between components to cause them to be aligned properly. This *alignment filler* is sometimes needed when the component is a record type or contains record types.

The packed representation for an array type differs from the unpacked representation if the type satisfies one of the following mutually exclusive requirements:

- The minimum size of the component type is less than 32 bits. In the packed representation, each component will occupy exactly the number of bits in its minimum size; this means that components might not start on storage-unit boundaries. If the last storage unit of a packed array of this type contains unused bits, the CDF will cause these bits, which are called *tail filler*, to be zeroed. This permits comparison and copy operations on the array to be performed efficiently using block operations.
- Alignment filler is used in the unpacked representation. In the packed representation, the alignment filler will be omitted; each component will still start on a storage unit-boundary.

If neither of the two situations above holds, the packed representation is the same as the unpacked representation.

If an array type has neither pragma Pack nor a length clause, the CDF uses the unpacked representation. If pragma Pack is applied to the array type, the CDF uses the packed representation. If a length clause is applied to the array type, the unpacked representation is used if it would result in a size less than or equal to that specified in the length clause; otherwise the packed representation is used. If the packed representation is also too large for the length clause, the length clause is rejected. If both pragma Pack and length clause are specified, the packed representation is used; an error will result if this representation is too large for the length clause.

Note that the length clause specifies an upper bound for the size of the array; a length that is less than the size of the unpacked representation will result in the use of the packed representation, even it if is smaller than the size given in the length clause.

Change of representation is supported for arrays. Hence, pragma Pack on a derived array type is honored, and length clauses on derived array types are permitted.

Record Types

In the absence of a record representation clause, a record type also has two basic representations: *unpacked* and *packed*.

In the unpacked representation, each record component starts on a storage-unit boundary, and alignment filler may be introduced between components to cause the components to be aligned properly. For example, an integer component will be aligned so that it starts on a longword boundary.

In the packed representation, a component whose size is less than 32 bits will occupy exactly the number of bits in its minimum size; such components might not start on storage unit boundaries. At present, alignment filler is not eliminated in the packed representation.

The criteria for selecting which representation to use are the same as described above for array types.

In either case, the CDF may lay out the record fields in a different order than that used in the type declaration for the record. This is done in an attempt to satisfy alignment requirements without introducing unnecessary alignment filler.

If a record representation clause is present, it may mention some or all of the fields in the record. Those fields that are mentioned in the clause will be laid out according to its rules; the remaining fields are then laid out according to the default algorithms, starting at the first storage unit past the last field mentioned in the clause. This is the case even if the clause leaves "holes" that are big enough to contain some of the fields not mentioned. In a discriminated record, this rule has an important consequence: if one of the discriminants is not mentioned in the clause, it will be placed *after* all of the fields in the largest variant part (as specified by the clause). Even though some of the variant parts may be smaller than others, constrained copies of the record selecting those variants will be as large as copies of the record with the largest variant.

A record representation clause cannot mention a field whose size is not known at compile time (this includes fields whose size depends on a discriminant).

IMPLEMENTATION-GENERATED NAMES

Implementation-Dependent Components

The LRM allows for the generation of names denoting implementation-dependent components in records. No such names are visible to the user for the M68020 Family CDF.

ADDRESS CLAUSES (LRM 13.5)

Address clauses can be applied to any object. Note that the address must be determinable at compile time, but it does not need to be Ada static (as defined in the LRM, Section 4.9). The 'Address attribute does not produce a compile-time static value. Addresses must be specified using constants and operators from package System.

Address clauses can be attached to a task entry even when the task entry is used for interrupt handling; however, in this case, the task entry *must* be available at the time of the interrupt. See the discussion on pragma Interrupt_Handler for additional information.

UNCHECKED PROGRAMMING

Unchecked Storage Deallocation (LRM 13.10.1)

Unchecked storage deallocation is implemented by the generic function Unchecked_Deallocation defined by the LRM. This procedure can be instantiated with an object type and its access type, resulting in a procedure that deallocates the object's storage. Objects of any type can be deallocated.

The storage reserved for the entire collection associated with an access type is reclaimed when the program exits the scope in which the access type is declared. Placing an access-type declaration within a block can be a useful implementation strategy when conservation of memory is necessary. Space on the free list is coalesced when objects are deallocated.

Erroneous use of dangling references may be detected in certain cases. When detected, the Storage_Error exception is raised. Deallocation of objects that were not created through allocation (that is, through Unchecked_Conversion) may also be detected in certain cases, also raising Storage_Error.

Unchecked Type Conversion (LRM 13.10.2)

Unchecked conversion is implemented by the generic function Unchecked_Conversion defined by the LRM. This function can be instantiated with *source* and *target* types, resulting in a function that converts source data values into target data values.

Unchecked conversion moves storage units from the source object to the target object sequentially, starting with the lowest address. Transfer continues until the source object is exhausted or the target object runs out of room. If the target is larger than the source, the remaining bits are undefined. Depending on the target-computer architecture, the result of conversions may be right- or left-aligned.

Restrictions on Unchecked Type Conversion

- The target type of an unchecked conversion cannot be an unconstrained array type or an unconstrained discriminated type without default discriminants.
- Internal consistency among components of the target type is not guaranteed. Discriminant components may contain illegal values or be inconsistent with the use of those discriminants elsewhere in the type representation.

INPUT/OUTPUT PACKAGES

The Ada language defines specifications for four I/O packages: Sequential_Io, Direct_Io, Text_Io, and Low_Level_Io. The following paragraphs explain the implementation-dependent characteristics of those four packages provided with the M68020/Bare CDF.

Sequential_Io. (LRM 14.2.2 and 14.2.3)

Because the M68020/Bare CDF does not support a file system, the implementation of Sequential_Io is quite simple. The Sequential_Io.Create and Sequential_Io.Open raise Io_Exception.Use_Error procedures unconditionally. The Sequential_Io.Is_Open function unconditionally returns False. All other procedures and functions exported by instantiations of Sequential_Io unconditionally raise Io_Exceptions.Status_Error.

Direct_Io. (LRM 14.2.4)

Because the M68020/Bare CDF does not support a file system, the implementation of Direct_Io is quite simple. The Direct_Io.Create and Direct_Io.Open procedures raise Io_Exception.Use_Error unconditionally. The Direct_Io.Is_Open function unconditionally returns False. All other procedures and functions exported by instantiations of Direct_Io unconditionally raise Io_Exceptions.Status_Error.

Low_Level_Io. (LRM 14.6)

Package Low_Level_Io can be used to obtain byte-oriented access to the address space of the MC68020 processor. The implementation-defined declarations for the package are:

```
type Device_Address is new System.Address;
type Signed_Byte    is new Integer range -128..127;
type Unsigned_Byte  is new Integer range 0..255;
```

The Send_Control and Receive_Control procedures are implemented so as to cause MC68020 bus cycles to user data space with the address specified. In the event of a Bus_Error during the bus cycle incurred by the call to either Send_Control or Receive_Control, the call returns with no indication. Data returned by Receive_Control in the face of a Bus_Error is not defined. Low_Level_Io is implemented using the Read_Memory and Write_Memory procedures found in package Runtime_Interface.Target_Dependent.

Text_Io. (LRM 14.3)

Specification of Package Text_Io (LRM 14.3.10)

The declaration of the type Count in Text_Io is:

```
type Count is range 0 .. 1_000_000_000;
```

The declaration of the subtype Field in Text_Io is:

```
subtype Field is Integer range 0 .. Integer'Last;
```

Unlike Direct_Io and Sequential_Io, the package Text_Io provides mechanisms for performing I/O to the Standard I/O device. The Mc68020_Bare CDF routines Standard_Input and Standard_Output to the system console via the Console_Io package. The implementation-defined aspects of Text_Io and the operation of several subprograms exported from Text_Io are described below.

Implementation-Dependent Characteristics.

The two implementation dependent types exported from Text_Io have the following implementation with the Mc68020_Bare CDF:

```
type    Count is range 0 .. 1_000_000_000;
subtype Field is Integer range 0 .. Integer'Last
```

The implementation-dependent line terminator is Ascii.Cr during input operations and the sequence Ascii.Cr, Ascii.Lf during output operations. The implementation dependent page terminator is Ascii.Ff.

File Management Operations

- *Create/Open*. If the file variable provided is either Standard_Input or Standard_Output, the exception Io_Exception.Status_Error is raised. Any other attempt results in Io_Exceptions.Use_Error.
- *Close*. If the file variable provided is either Standard_Input or Standard_Output, the Close is performed. Any other attempt results in Io_Exceptions.Status_Error.
- *Delete/Reset*. If the file variable provided is either Standard_Input or Standard_Output then Io_Exceptions.Use_Error is raised. Any other attempt results in Io_Exceptions.Status_Error.
- *Name/Form*. If the file variable provided is either Standard_Input or Standard_Output, the null string is returned. Any other attempt results in Io_Exception.Status_Error.
- *Get and Put*. The operation of Get and Put is as described in the LRM. Data written using Put and Put_Line is not interpreted in any fashion. Data written using Put_Line is followed by the line terminator sequence Ascii.Cr, Ascii.Lf. Data read using Get and Get_Line is not interpreted except that the line terminator, Ascii.Cr, and the page terminator, Ascii.Ff, are removed from the input stream. Data read from Standard_Input is echoed to Standard_Output except that the line terminator Ascii.Cr is echoed as the sequence Ascii.Cr, Ascii.Lf and the remaining control characters Ascii.Nul .. Ascii.Ff, Ascii.So .. Ascii.Us, and Ascii.Del are not echoed.
- *Other properties of Text_Io*. Text_Io output operations are synchronized by the implementation of Text_Io with respect to Console_Io. This typically ensures that the entire Item passed to any of the overloaded Put procedures reaches the terminal atomically. If, however, the Item cannot be placed on a single line because of bounded line lengths then only those portions of the Item that fit on a single line will reach the terminal atomically. The line terminator sequence (output for calls to Put_Line) is not necessarily synchronous with the corresponding Item.

OTHER IMPLEMENTATION-DEPENDENT FEATURES

Machine Code (LRM 13.8)

Machine-code insertions are not supported at this time.